Pipelining
Putting it all together

Virtual address <64>

Virtual page number <51> Page offset <13>

TLB tag compare address <43> TLB index <8>

L1 cache index <7> Block offset <6>

TLB tag <43> TLB data <28>

L1 cache tag <43> L1 data <512>

Physical address <41>

L2 tag compare address <19> L2 cache index <16> Block offset <6>

L2 cache tag <19> L2 data <512>

To CPU

To L1 cache or CPU
Another example

• Virtual Memory Address width: 32 bits Page size: 1 K bytes Single level page table Physical Memory 32 bit physical address space Cache Block size: 16 bytes Cache size: 1 K bytes Associativity: Direct mapped Translation Lookaside Buffer Number of translations: 64 Associativity: Direct mapped
## 2-Level TLB Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual addr</td>
<td>48 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td>Physical addr</td>
<td>44 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td>Page size</td>
<td>4KB, 2/4MB</td>
<td>4KB, 2/4MB</td>
</tr>
</tbody>
</table>
| L1 TLB (per core)   | L1 I-TLB: 128 entries for small pages, 7 per thread (2×) for large pages  
L1 D-TLB: 64 entries for small pages, 32 for large pages  
Both 4-way, LRU replacement | L1 I-TLB: 48 entries  
L1 D-TLB: 48 entries  
Both fully associative, LRU replacement |
| L2 TLB (per core)   | Single L2 TLB: 512 entries  
4-way, LRU replacement | L2 I-TLB: 512 entries  
L2 D-TLB: 512 entries  
Both 4-way, round-robin LRU |
| TLB misses          | Handled in hardware                               | Handled in hardware                             |
# 3-Level Cache Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 caches</strong></td>
<td>L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a</td>
<td>L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles</td>
</tr>
<tr>
<td>(per core)</td>
<td>L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>L2 unified</strong></td>
<td>256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
<td>512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a</td>
</tr>
<tr>
<td><strong>cache</strong> (per core)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>L3 unified</strong></td>
<td>8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a</td>
<td>2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles</td>
</tr>
<tr>
<td><strong>cache (shared)</strong>*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

n/a: data not available
Pipelining: Natural Phenomenon

Laundry Example:

Ann, Brian, Ming, Ashok each has one load of clothes to wash, dry, and fold

Washer takes 30 minutes

Dryer takes 40 minutes

“Folder” takes 20 minutes
• Sequential laundry takes 6 hours for 4 loads
Pipelined Laundry -- Start work ASAP

Tasks:
- A
- B
- C
- D

Orders:
- Task A
- Task B
- Task C
- Task D

Time:
- 6 PM
- 7
- 8
- 9
- 10
- 11
- Midnight

Activities:
- 30
- 40
- 40
- 40
- 40
- 20
Pipelined Laundry -- Start work ASAP

- Pipelined laundry takes 3.5 hours for 4 loads
- Total time to wash longer or shorter?
- Total wait time longer or shorter?
- What’s the utilization of the dryer?
- What’s the utilization of the folding table?
- What determines the throughput?
Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number of pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup

Diagram:

- Tasks: A, B, C, D
- Time: 6 PM, 7, 8, 9
- Time to process:
  - Task A: 30
  - Task B: 40
  - Task C: 40
  - Task D: 20

Graph shows the order of tasks and their processing times.
Review -- Instruction Latencies

**Single-Cycle CPU**

- Load: Ifetch, Reg/Dec, Exec, Mem, Wr

**Multiple Cycle CPU**

- Load: Ifetch, Reg/Dec, Exec, Mem, Wr
  - Cycle 1
  - Cycle 2
  - Cycle 3
  - Cycle 4
  - Cycle 5

- Add: Ifetch, Reg/Dec, Exec, Wr
# Instruction Latencies and Throughput

- **Single-Cycle CPU**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>周期</th>
<th>操作</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
</tr>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
</tr>
</tbody>
</table>

- **Multiple Cycle CPU**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
</tr>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
</tr>
</tbody>
</table>

- **Pipelined CPU**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
</tr>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
</tr>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
</tr>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
</tr>
</tbody>
</table>

| Load    | Ifetch  | Reg/Dec | Exec    | Mem     | Wr      | Ifetch  | Reg/Dec | Exec    |
| Load    | Ifetch  | Reg/Dec | Exec    | Mem     | Wr      | Ifetch  | Reg/Dec | Exec    |
| Load    | Ifetch  | Reg/Dec | Exec    | Mem     | Wr      | Ifetch  | Reg/Dec | Exec    |
Execution in a Pipelined Datapath

steady state
Speed up calculation

• Assuming the stages are perfectly balanced

• \( \text{Time between instructions (pipelined)} = \frac{\text{Time between instructions (nonpipelined)}}{\text{Number of pipe stages}} \)
Pipelining Example 1

• A processor that takes 5ns to execute an instruction is pipelined into 5 equal stages. The latch between each stage has a delay of 0.25 ns.
  – What is the minimum clock cycle time of the pipelined processor?
    • $5/5 + 0.25 = 1.25$ ns
  – What is the maximum speedup that can be achieved by this pipelined processor? (compared to the original processor)
    • $4\times (1 \text{ instr every 1.25ns vs 1 instr every 5 ns})$
  – Can we have much deeper pipelining?
    • If we divide into 10 stages, the clock will be 0.75 ns and the speedup will at most $6.7\times$, diminishing return!
Pipelining Example 2

A. A non-pipelined processor takes 5ns to execute an instruction. If I want the processor to be clockable at 2GHz, how many stages should I pipeline this processor into if each latch has a 0.25ns delay?

B. How many stages if I want to clock the processor at 5GHz?

C. What is the maximum speedup that can be achieved by the pipelined processor running at 2GHz? (Compared to the original single cycle processor)

D. What is the average latency of an instruction (belonging to a 23-instruction program) for the pipelined processor?

E. What is the best case speedup and what is the worst case slowdown?
Hazards

- Situations that prevent starting the next instruction in the next cycle

- Structure hazards
  - A required resource is busy

- Data hazard
  - Need to wait for previous instruction to complete its data read/write

- Control hazard
  - Deciding on control action depends on previous instruction
Data Hazards

- When a result is needed in the pipeline before it is available, a “data hazard” occurs.

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td>R2 Available</td>
<td>R2 Needed</td>
<td>IM</td>
<td>Reg</td>
</tr>
<tr>
<td>sub $2, $1, $3</td>
<td>and $12, $2, $5</td>
<td>or $13, $6, $2</td>
<td>add $14, $2, $2</td>
<td>sw $15, 100($2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hazards! Value in register hasn't been updated</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No hazard here
add $10, $1, $2
add $11, $8, $7
lw $8, 52($3)
add $3, $10, $11

Write Back

And this should be value from memory (which hasn’t even been loaded yet).

This should have been “92”
Branch Hazards

beq $2, $1, here

add ...

sub ...

lw ...

here: lw ...

These instructions shouldn’t be executed!

Finally, the right instruction
Dealing With Data Hazards

Transparent register file eliminates one hazard.

Use latches rather than flip-flops in Reg file

• First half-cycle of cycle 5: register 2 loaded

• Second half-cycle: new value is read into pipeline state

sub $2, $1, $3

and $12, $6, $5

or $13, $6, $8

add $14, $2, $2
Dealing with Data Hazards in Software

Insert enough no-ops (or other instructions that don’t use register 2) so that data hazard doesn’t occur,
Handling Data Hazards in Hardware
Stall the pipeline

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
Pipeline Stalls

• To insure proper pipeline execution in light of register dependences, we must:
  – Detect the hazard
  – Stall the pipeline
    • prevent the IF and ID stages from making progress
    • insert “no-ops” into later stages
Register Scoreboard – Tracking Operand Availability

- Add valid bit to each register in the register file
- Hardware clears valid bit when an instruction that writes the register issues (leaves decode/register read stage)
- Hardware sets valid bit when an instruction that writes the register completes
- Instructions not allowed to issue if any of their source registers are invalid
Stalling the Pipeline

• Prevent the IF and ID stages from proceeding
  – don’t write the PC (PCWrite = 0)
  – don’t rewrite IF/ID register (IF/IDWrite = 0)
• Insert “nops”
  – set all control signals propagating to EX/MEM/WB to zero
Reducing Data Hazards Through Forwarding

We could avoid stalling if we could get the ALU output from “add” to ALU input for the “or”
Eliminating Data Hazards via Forwarding

sub $2, $1, $3
and $6, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Forwarding Paths

b. With forwarding
Forwarding Conditions

• EX hazard
  – if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    \[\text{ForwardA} = 10\]
  – if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    \[\text{ForwardB} = 10\]

• MEM hazard
  – if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    \[\text{ForwardA} = 01\]
  – if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    \[\text{ForwardB} = 01\]
Double Data Hazard

• Consider the sequence:
  - add $1, $1, $2
  - add $1, $1, $3
  - add $1, $1, $4

• Both hazards occur
  - Want to use the most recent

• Revise MEM hazard condition
  - Only fwd if EX hazard condition isn’t true
Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
Does Forwarding Eliminate All Data Hazards

lw $2, 10($1)

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
Does Forwarding Eliminate All Data Hazards

lw $2, 10($1)
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Does Forwarding Eliminate All Data Hazards

lw $2, 10($1)

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
Does Forwarding Eliminate All Data Hazards

lw $2, 10($1)

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)
Try this one...

Show stalls and forwarding for this code

```assembly
add $3, $2, $1
lw $4, 100($3)
and $6, $4, $3
sub $7, $6, $2
```
Code Scheduling to Avoid Stalls

• Reorder code to avoid use of load result in the next instruction

• C code for $A = B + E; C = B + F$;
Next lecture

• Control hazards

• References for the current lecture
  – Chapter 4.5, 4.6, 4.7, 4.8